Point clouds in PostgreSQL: store and publish

This talk discusses point clouds, the Pointcloud extension for storing point clouds in PostgreSQL, and the LOPoCS lightweight server for streaming point clouds on the web.

Éric Lemoine

Developer @ Oslandia

FOSS4G developer and enthusiast since 2007

eric.lemoine@oslandia.com
@elemoine
@erilem

My name is Éric Lemoine. I work at Oslandia. And I've been working in the FOSS4G field since 2007.

Oslandia

Oslandia provides service on open-source software

- GIS
- 3D
- DATA

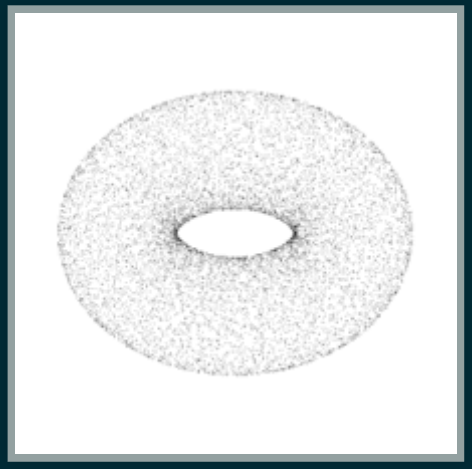
Oslandia is an open-source company working on GIS, 3D and Data Science. QGIS, PostGIS and the iTowns 3D WebGL framework are examples of software components we are working on.

Point clouds!

Let's talk about point clouds in general first!

Point clouds

« A point cloud is a set of data points in space. »



source: wikipedia

A point cloud is just a set of data points in space. Nothing more. Point clouds provide a way to represent objects of our environment. A church and streets around it in the previous slide, and a donut here.

Point clouds

- Generally produced by 3D scanners (LiDAR)
- Can also be created using Photogrammetry

What can produce point clouds? Point clouds are generally produced by 3D scanner. This is the LiDAR (Light Detection And Ranging) technology. Point clouds can also be produced using photogrammetry techniques (through homolog points).

LiDAR

Terrestrial, Airborne, Mobile, Unmanned

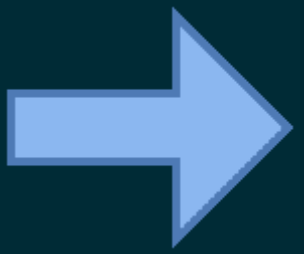
There are several types of LiDAR acquisitions: Terrestrial (fixed tripods), Airbone (planes or helicopters), Mobile (Google Car like), and Unmanned (drones).

Many applications!

- Create Digital Elevation Models (DEMs)
- Create 3D models
- Detect objects and obstacles
- etc.

Point clouds have a wide range of applications. Examples include creating Digital Evelation Models, Digital Surface Models, 3D models, and detecting objects and obstacles. Autonomous cars use LiDAR! For the creation of 3D models, 3D surfaces are derived from point clouds.

Point clouds in PostgreSQL

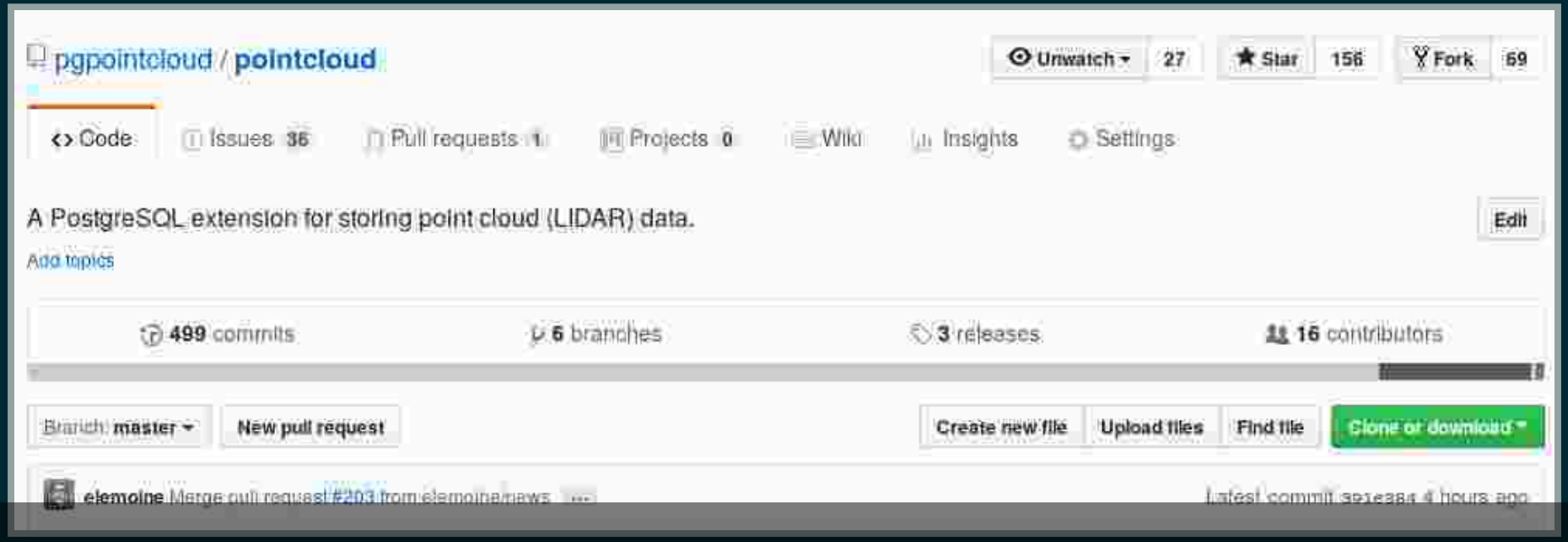


Now let's discuss the Pointcloud extension for PostgreSQL.

Pointcloud

"PostgreSQL extension for storing point cloud data"

https://github.com/pgpointcloud/pointcloud



The Pointcloud extension allows storing point cloud data in PostgreSQL databases. Pointcloud is open-source and available on GitHub. It's easy to build and install, and it's well documented.

Pointcloud

- Initially developed by Paul Ramsey (funded by Natural Resources Canada)
- Now developed and maintained by Oslandia and IGN

The initial development of Pointcloud was funded by Natural Resources Canada, and done by Paul Ramsey, one of the main PostGIS developers. It is currently developed and maintained by Oslandia and IGN (mostly).

Goals

- Storing LiDAR data in PostgreSQL
- Leveraging that data for analysis in PostGIS

Storing LiDAR data in PostgreSQL enables all sort of analysis, by using PostGIS and Pointcloud together. For example determining all the points that are within a polygon is both a very easy and very fast operation.

Why not use PostGIS?

```
     Column       |       Type
------------------+------------------
 id               | integer
 geom             | geometry(PointZ)
 intensity        | double precision
 returnnumber     | double precision
 numberofreturns  | double precision
 classification   | double precision
 scananglerank    | double precision
 red              | double precision
 green            | double precision
 blue             | double precision
```

By the way, why not using PostGIS instead of creating a specific extension? PostGIS has a PointZ geometry type that could be used, hasn't it?

Why not use PostGIS?

- One point per row means billions of rows
- Does not work!

Because point clouds may have billions of points, which would mean billions of database rows, which wouldn't work.

Patches of points

- Organize the points into patches
- → Millions of rows instead of billions

```
 Column |    Type
--------+------------
 id     | integer
 pa     | pcpatch(1)
```

For that reason Pointcloud organizes points into patches. A patch typically includes several hundreds or several thousands points, which translates into millions of rows rather than billions of rows. This is still big, but manageable.

Two types

- `PcPoint(pcid)`
- `PcPatch(pcid)`

Pointcloud actually defines two new types: PcPoint and PcPatch. PcPatches are collections of PcPoints. PcPoints are packings of point dimensions (X, Y, Z, …). Dimensions are packed in byte arrays.

Use Pointcloud

```
CREATE EXTENSION pointcloud;
CREATE EXTENSION postgis;            -- optional
CREATE EXTENSION pointcloud_postgis; -- optional
```

Enabling Pointcloud in a database is done the way as enabling PostGIS.

Use Pointcloud

```
 Schema |        Name         |   Type
--------+---------------------+----------
 public | geography_columns   | view
 public | geometry_columns    | view
 public | pointcloud_columns  | view
 public | pointcloud_formats  | table
 public | raster_columns      | view
 public | raster_overviews    | view
 public | spatial_ref_sys     | table
```

After enabling Pointcloud in a database the pointcloud_colums view and the pointcloud_formats table are added to the database. The poincloud_colums view includes information about all the PcPoint and PcPatch columns that exist in the database. The pointcloud_formats table includes XML documents that define how dimensions are encoded in PcPoints.

Schema

```
 pcid | srid |                                  schema
------+------+-------------------------------------------------------------------------------
    1 | 4326 | <?xml version="1.0" encoding="UTF-8"?>                                       +
      |      | <pc:PointCloudSchema xmlns:pc="http://pointcloud.org/schemas/PC/1.1"         +
      |      |     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">                   +
      |      |   <pc:dimension>                                                             +
      |      |     <pc:position>1</pc:position>                                             +
      |      |     <pc:size>4</pc:size>                                                     +
      |      |     <pc:description>X coordinate as a long integer. You must use the         +
      |      |                     scale and offset information of the header to            +
      |      |                     determine the double value.</pc:description>             +
      |      |     <pc:name>X</pc:name>                                                     +
      |      |     <pc:interpretation>int32_t</pc:interpretation>                           +
      |      |     <pc:scale>0.01</pc:scale>                                                +
      |      |   </pc:dimension>                                                            +
      |      |   <pc:dimension>                                                             +
      |      |     <pc:position>2</pc:position>                                             +
      |      |     <pc:size>4</pc:size>                                                     +
      |      |     <pc:description>Y coordinate as a long integer. You must use the         +
      |      |                     scale and offset information of the header to            +
      |      |                     determine the double value.</pc:description>             +
      |      |     <pc:name>Y</pc:name>                                                     +
      |      |     <pc:interpretation>int32_t</pc:interpretation>                           +
      |      |     <pc:scale>0.01</pc:scale>                                                +
      |      |   </pc:dimension>                                                            +
      |      |   <pc:dimension>                                                             +
      |      |     <pc:position>3</pc:position>                                             +
      |      |     <pc:size>4</pc:size>                                                     +
      |      |     <pc:description>Z coordinate as a long integer. You must use the         +
```

This is an example of an PointCloudSchema XML document.

```
SELECT pa FROM patches LIMIT 1;
------------------------------------------------------------------
 0101000000020000000900000002100000000400000060CEFFFFBC9A78560000
(1 row)
```

An SQL query that selects a patch (PcPatch) returns a sort of WKB (Well Known Binary) string representing the patch.

```
SELECT PC_AsText(pa) FROM patches LIMIT 1;
------------------------------------------------------------------
 {"pcid":1,"pts":[[-126.99,45.01,1,0],[-126.98,45.02,2,0],[-126.9
(1 row)
```

The Pointcloud extension provides functions for manipulating points and patches. For example the `PC_AsText` function returns a JSON representation of patches.

Working with real data

Let's look at how real point cloud data can be inserted into a PostgreSQL database?

PDAL

https://www.pdal.io/

PDAL can be used for that!

Load data using PDAL

```
{
  "pipeline": [
    {
        "type": "readers.las",
        "filename":"inrap.las"
    },
    {
        "type": "filters.chipper",
        "capacity": "400"
    },
    {
        "type":"writers.pgpointcloud",
        "connection":"dbname=lopocs host=localhost user=lopocs",
        "schema": "public",
        "table":"inrap",
        "compression":"none",
        "srid":"3946",
        "overwrite":"true",
        "column": "points",
        "scale_x": "0.01",
        "scale_y": "0.01",
        "scale_z": "0.01",
        "offset_x": "831587.0631",
        "offset_y": "6287650.923",
        "offset_z": "30.921565055000002"
    }
  ]
}
```

This creates a PDAL pipeline whose source is a LAS file and sink is a Pointcloud database table. The filter in between the source and the sink is a so-called "chipper" filter. The "chipper" filter is responsible for creating patches of points – 400-point patches here.

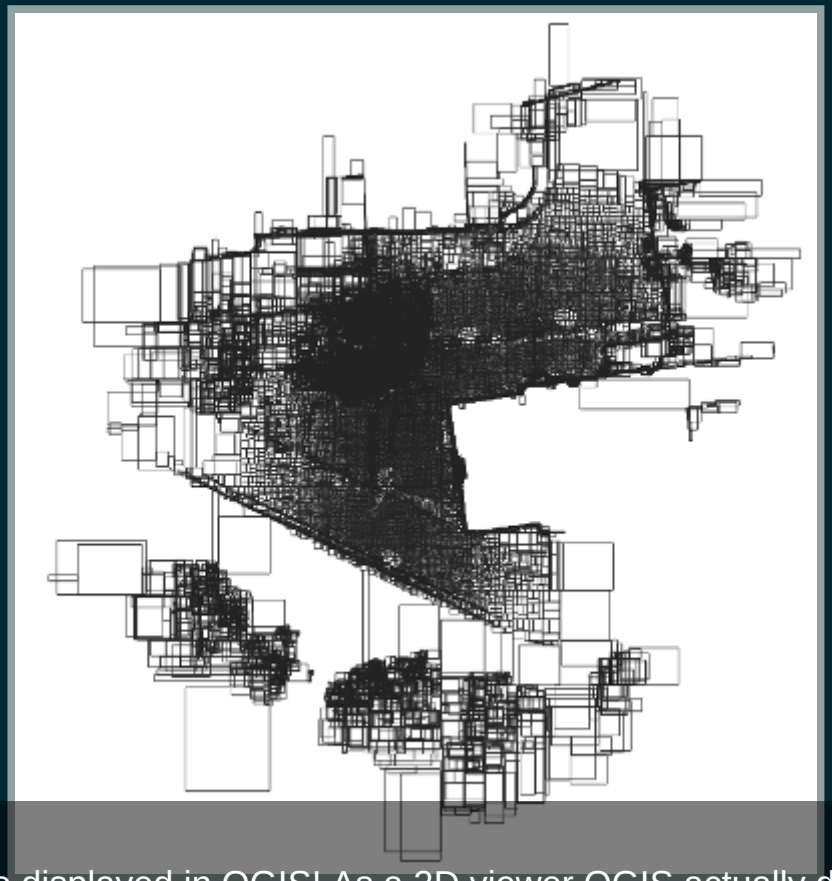
```
SELECT count(*) num_patches,
       sum(PC_NumPoints(points)) num_points
FROM inrap;

 num_patches | num_points
-------------+------------
       45952 |   18380597
(1 row)
```

The point cloud has been loaded into PostgreSQL. We can start throwing some SQL at it! The above SQL query just counts the total number of patches and points.

Visualize in QGIS

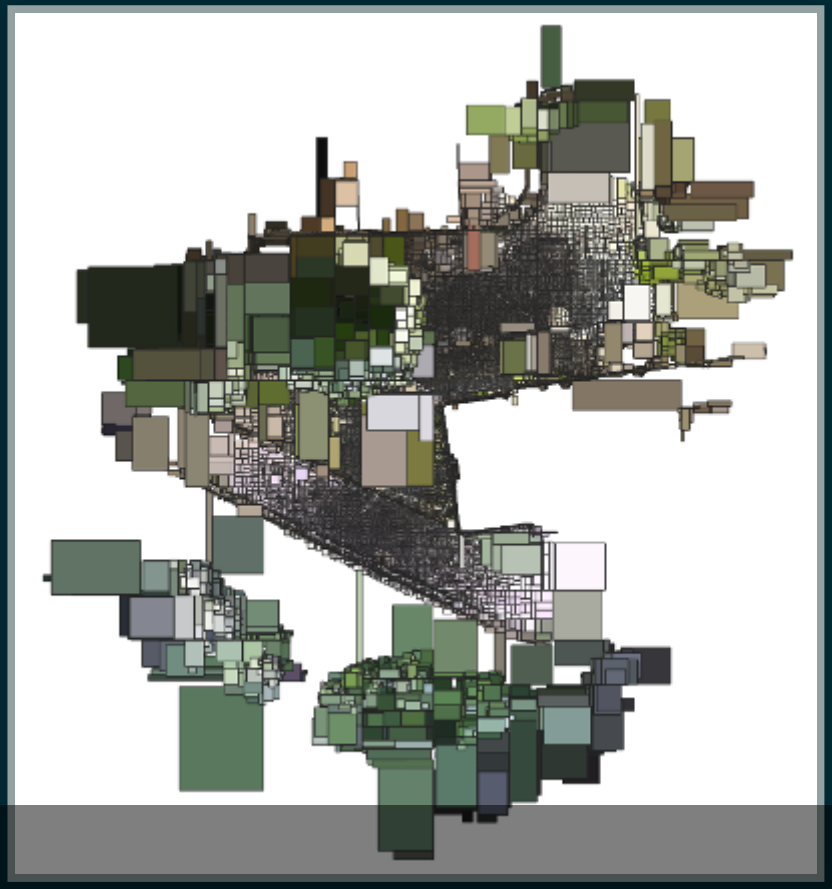
```
SELECT id, points FROM inrap
```



Point clouds in PostgreSQL can be displayed in QGIS! As a 2D viewer QGIS actually displays the 2D bounds (X/Y bounds) of patches. This is actually very useful for testing and debugging.

Visualize in QGIS

```
SELECT id, points, PC_PatchAvg(points, 'red') || ',' ||
                   PC_PatchAvg(points, 'green') || ',' ||
                   PC_PatchAvg(points, 'blue') || ',255' color FROM inrap;
```



We can even add some colors!

v1.1.0

Released the 2018-04-31

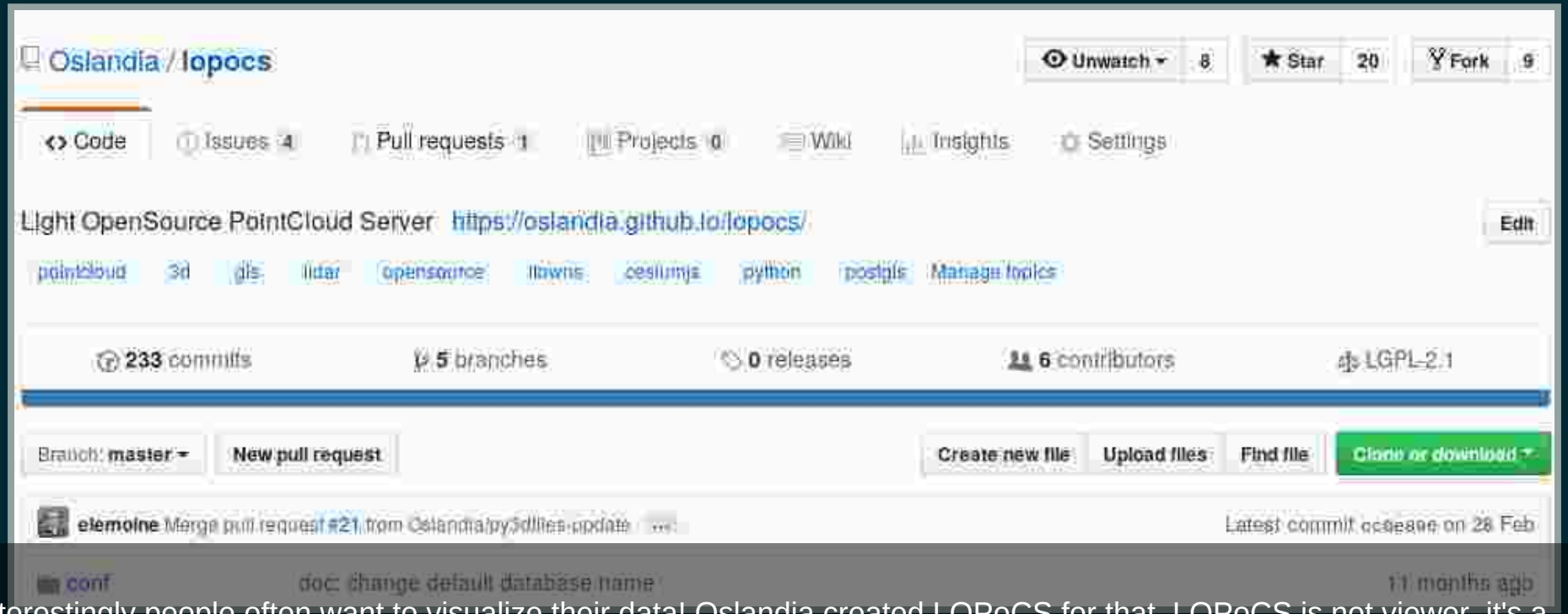
New functions include:

- `PC_Patch{Avg,Max,Min}(p pcpatch, dimname text)`
- `PC_Range(p pcpatch, start int4, n int4)`
- `PC_SetPCId(p pcpatch, pcid int4, def float8 default 0.0)`
- `PC_Transform(p pcpatch, pcid int4, def float8 default 0.0)`
- `PC_BoundingDiagonalAsBinary(p pcpatch)`

The Pointcloud extension includes many PcPoint and PcPatch manipulation functions. The 1.1.0 version, which was released the 2018-04-31, brings new functions that are useful both for analysis and visualization.

"Light OpenSource PointCloud Server"

https://github.com/Oslandia/lopocs



Interestingly people often want to visualize their data! Oslandia created LOPoCS for that. LOPoCS is not viewer, it's a light open-source pointcloud streaming server.

LOPoCS

- Streams point cloud data stored in PostgreSQL
- Supports multiple streaming protocols (Greyhound and 3D Tiles currently supported)
- → works with Potree, Cesium, iTowns

LOPoCS is able to stream point cloud data stored in PostgreSQL/Pointcloud. LOPoCS implements existing protocols. The Greyhound and 3D Tiles protocols are currently supported. This makes LOPoCS works with various point cloud web viewers, including Potree, Cesium and iTowns.

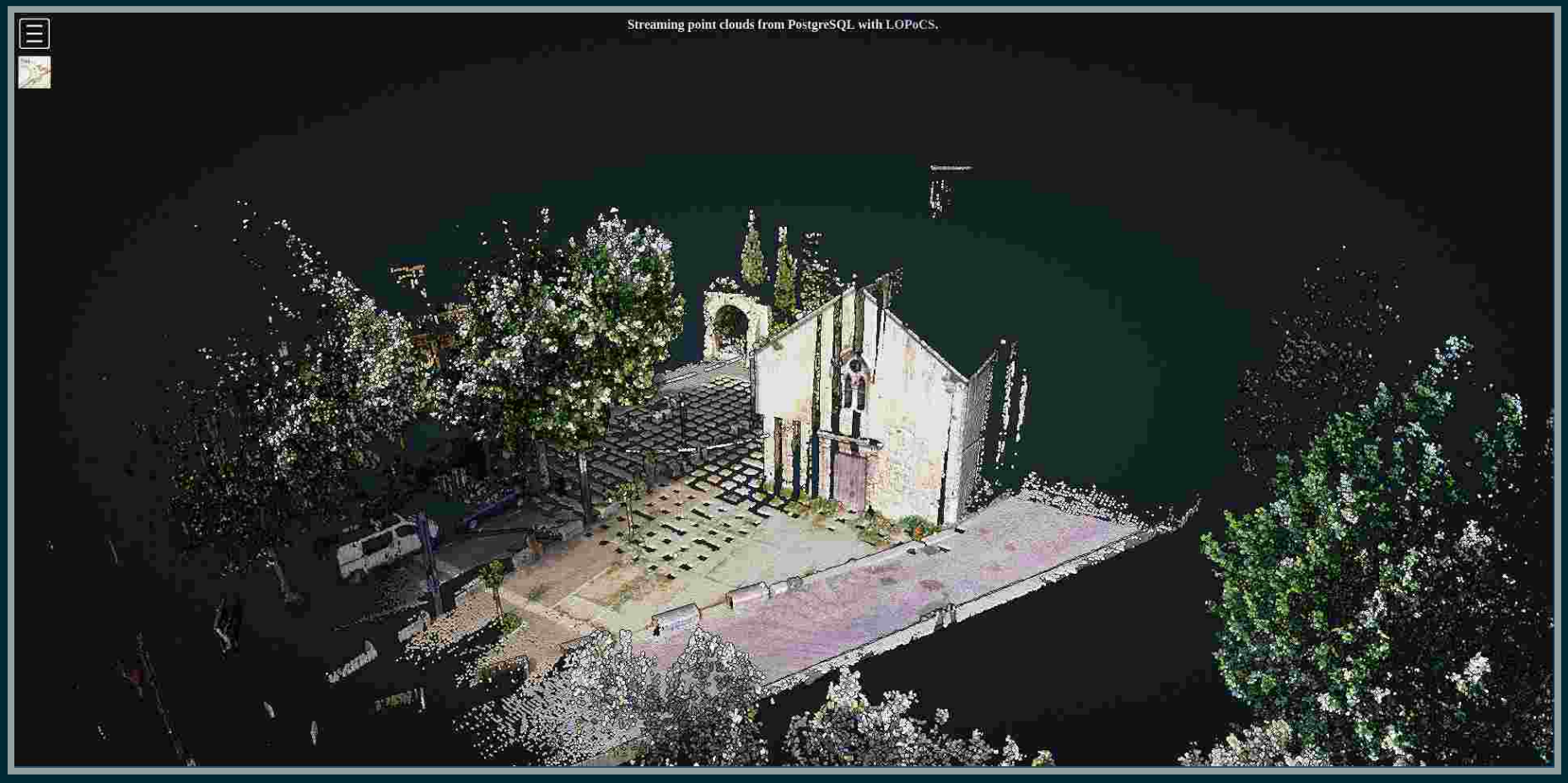
Greyhound in a nutshell

Greyhound is a dynamic point cloud server architecture that performs progressive level-of-detail streaming of indexed resources on-demand

https://greyhound.io/

Greyhound is a point cloud data streaming protocol. It was created by Howard Butler and Connor Maning from Hubo, Inc. It is the protocol used by the Greyhound server.

Potree/Greyhound



This is a Potree application displaying a point cloud streamed by LOPoCS.

3D Tiles in a nutshell

Specification for streaming massive heterogeneous 3D geospatial datasets

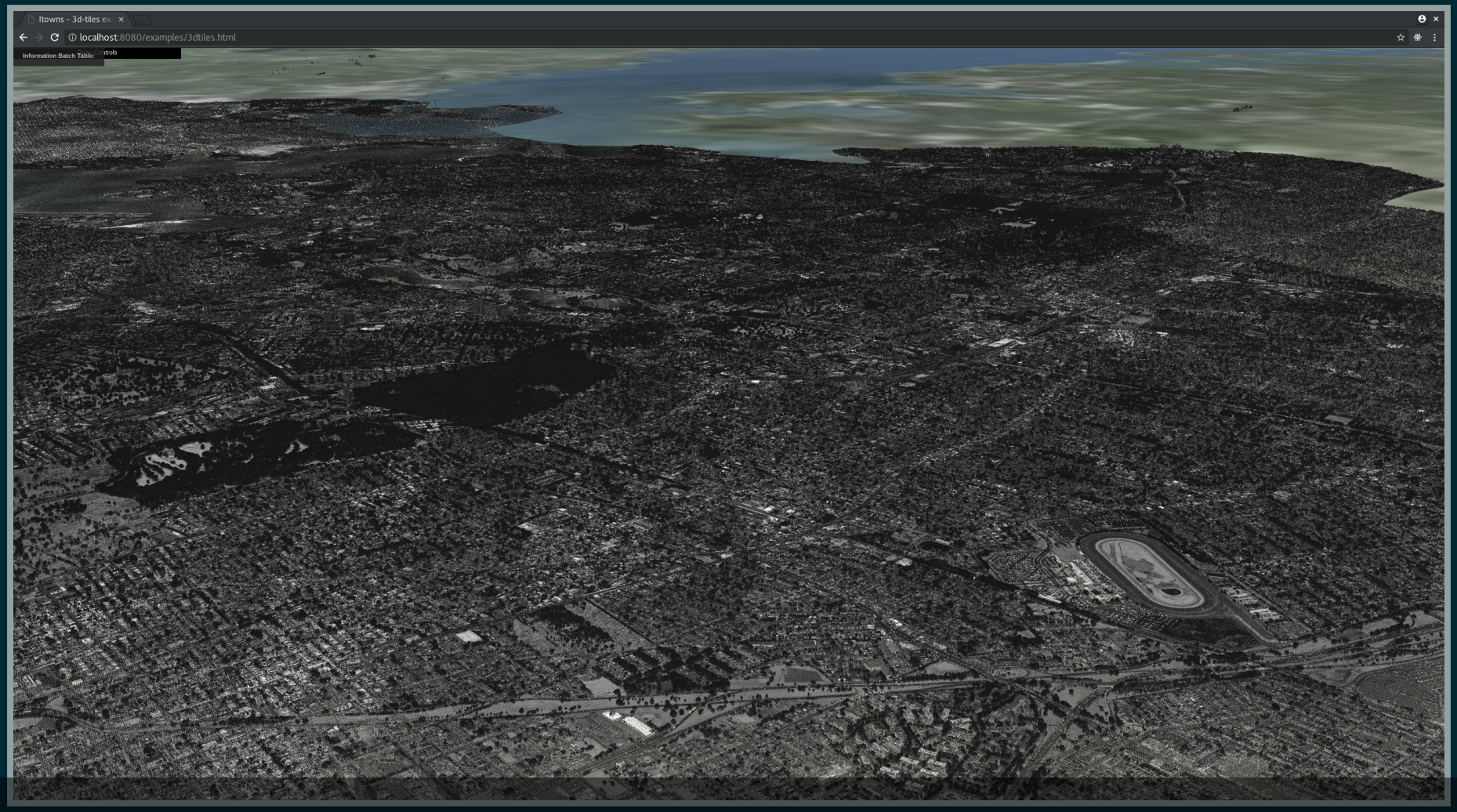
https://github.com/AnalyticalGraphicsInc/3d-tiles

3D Tiles a specification for streaming 3D content. Is is not specific to point cloud data. It can be used to stream building, trees, point clouds and vector data. It was created by the Cesium team, and has now entered the OGC Community Standard process.

Cesium/3D Tiles

This is a Cesium application displaying a point cloud streamed by LOPoCS.

iTowns/3D Tiles



This is an iTowns application displaying a point cloud streamed by LOPoCS.

Motivation

- Stream point cloud data directly from Postgres
- No export/indexing step required
- Nice for pre-visualization and prototyping

The motivation is to be able to stream point cloud data directly from PostgreSQL, without having to export and index the data outside the database. In particular this is useful for pre-visualization and prototyping. For serious visualizations exporting the data from indexed files (3D Tiles for example) will always lead to better performance.

Technologies

- Language: Python 3
- Web framework: Flask
- Main libraries: py3dtiles, lazperf

LOPoCS is written in Python 3. It uses the Flask web framework. It is based on the py3dtiles library (for 3D Tiles) and the laz-perf library (for Greyhound).

LOPoCS Future

- Improve the selection of points
- Make rendering as good and performant as possible

LOPoCS is still a young project. It still required work to improve the selection of points, and provide for a better rendering. We hope to make LOPoCS better and attract more users and developers in the future.

Thanks!